Computer Architectures
Session 3

Pipelined RISC-V processor with control-hazard handling

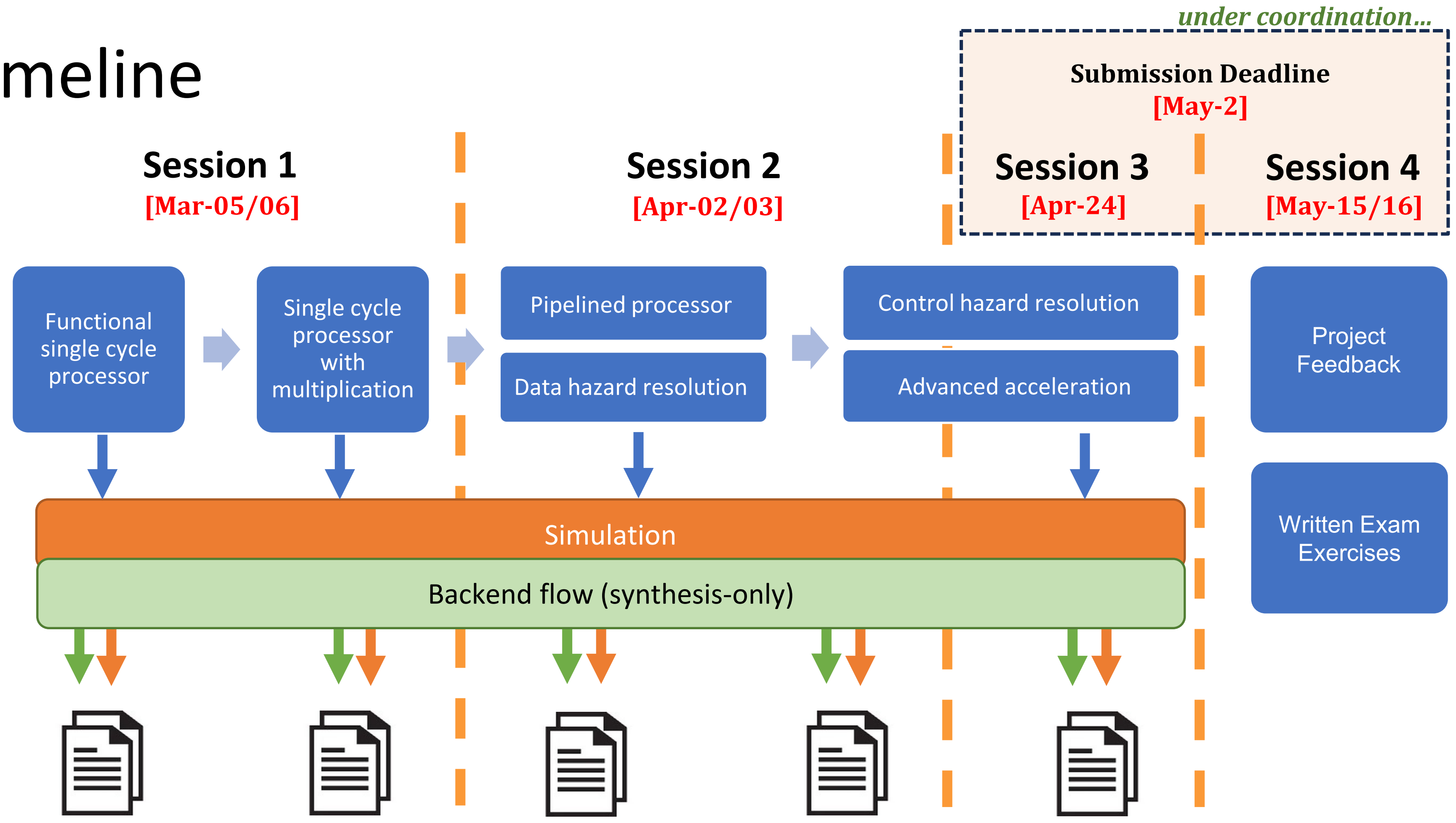
&

Advanced acceleration

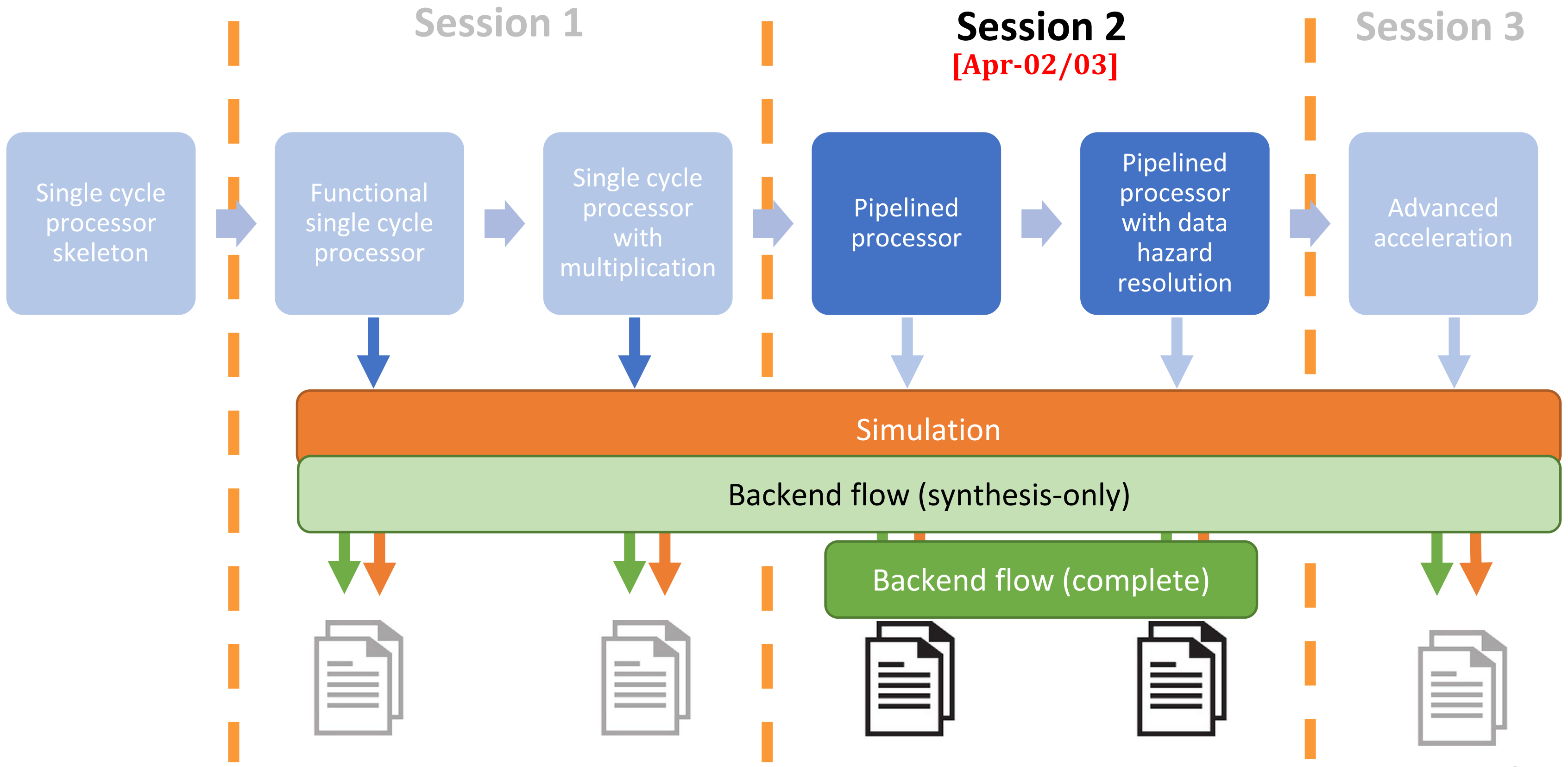
TAs :

Jun Yin (jun.yin@kuleuven.be)
Yuanyang Guo (yuanyang.guo@imec.be)
Xiaoling Yi (xiaoling.yi@kuleuven.be)
Yunzhu Chen (yunzhu.chen@imec.be)

Timeline


under coordination...
Submission Deadline
[May-2]
Session 1
[Mar-05/06]
Session 2
[Apr-02/03]
Session 3
[Apr-24]
Session 4
[May-15/16]
Functional single cycle processor
Single cycle processor with multiplication
Pipelined processor
Data hazard resolution
Control hazard resolution
Advanced acceleration
Project Feedback
Written Exam Exercises
Simulation
Backend flow (synthesis-only)

Last session recap


Session 1
Session 2
[Apr-02/03]
Session 3
Single cycle processor skeleton
Functional single cycle processor
Single cycle processor with multiplication
Pipelined processor
Pipelined processor with data hazard resolution
Advanced acceleration
Simulation
Backend flow (synthesis-only)
Backend flow (complete)

Last session recap

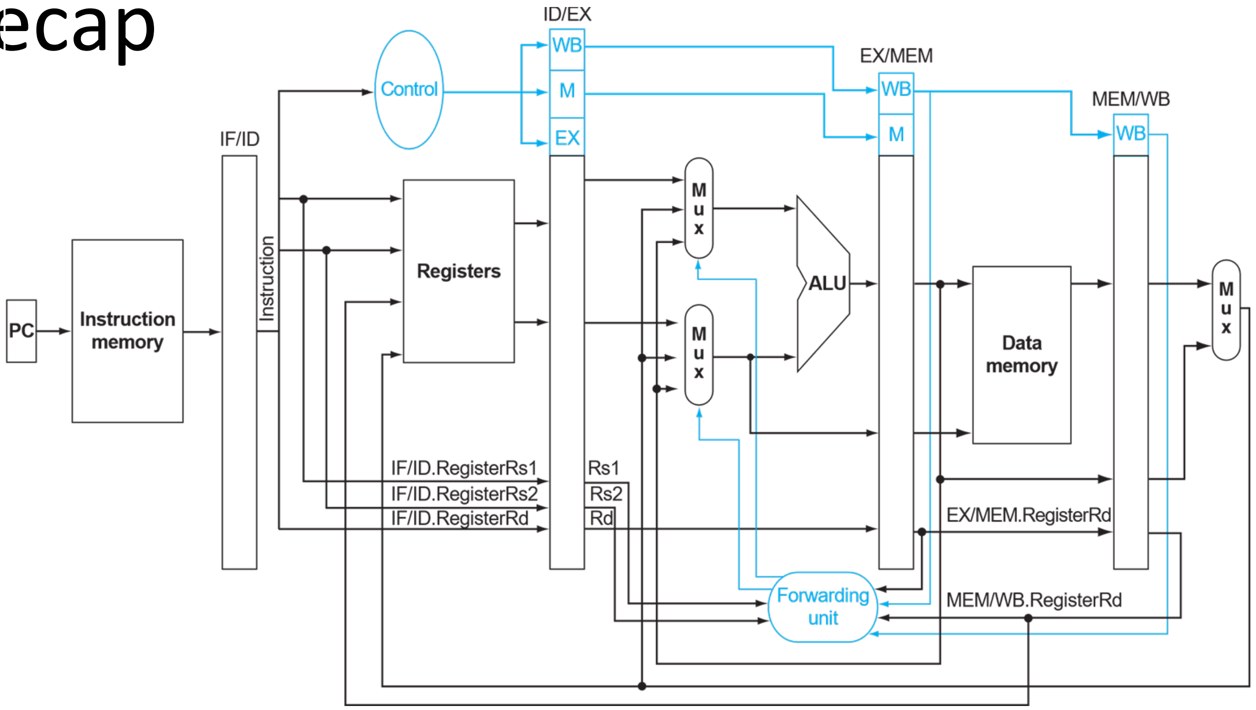
Pipelined Processor

✓ Mult2

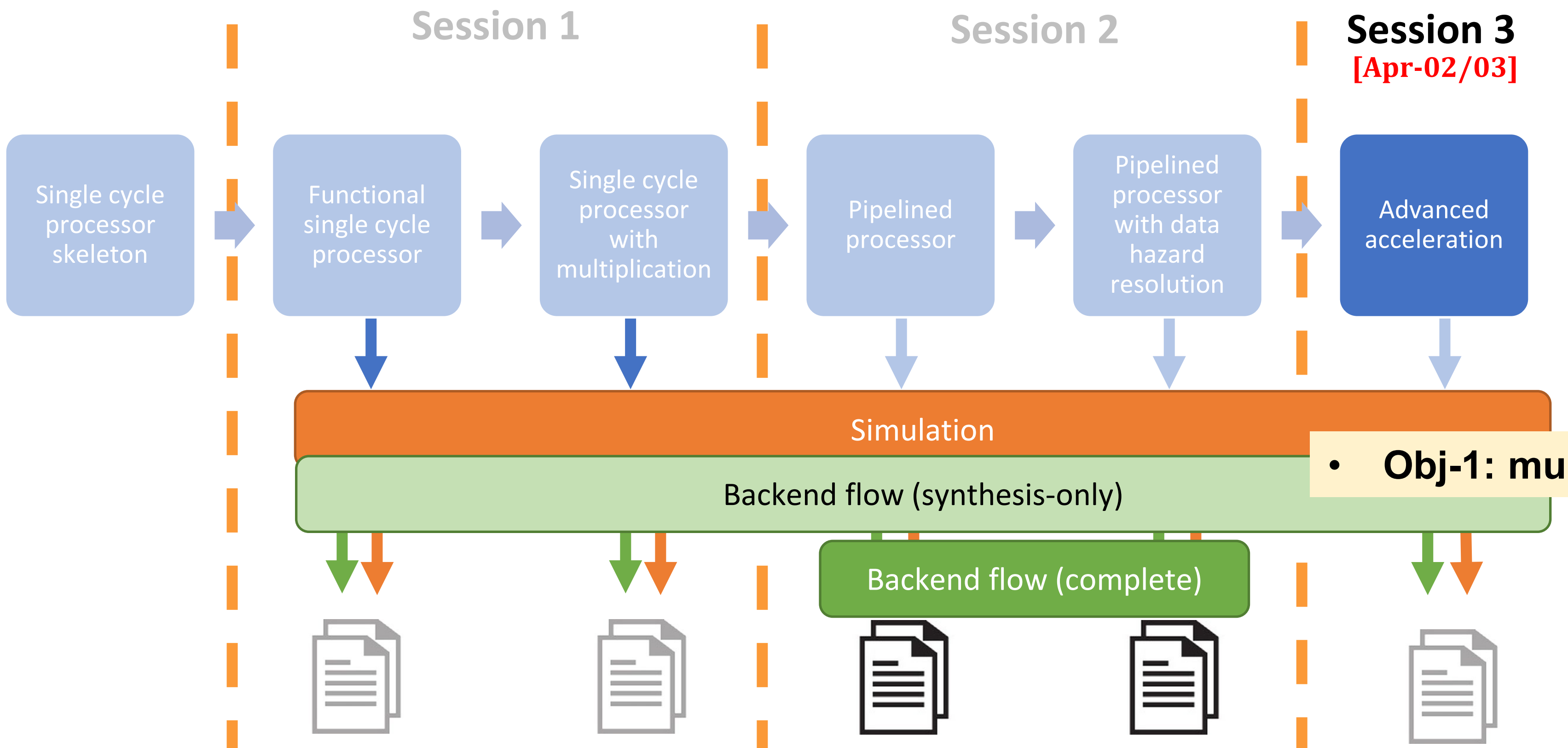
✓ Mult3

✓ Backend complete

Prerequisite for this session!



Today’s session: Advanced acceleration


Session 1
Session 2
Session 3
[Apr-02/03]
Single cycle processor skeleton
Functional single cycle processor
Single cycle processor with multiplication
Pipelined processor
Pipelined processor with data hazard resolution
Advanced acceleration
Simulation
Backend flow (synthesis-only)
Backend flow (complete)

- Obj-1: mult4

Obj-1: mult4

- Matrix-matrix multiplication (**MULT4**)
  - Input matrices stored in **mult4_dmem_content.txt**
  - Methods

```
i.  Add more RTL modules
ii. Modify the mult4_imem
    program
iii.or both …
```

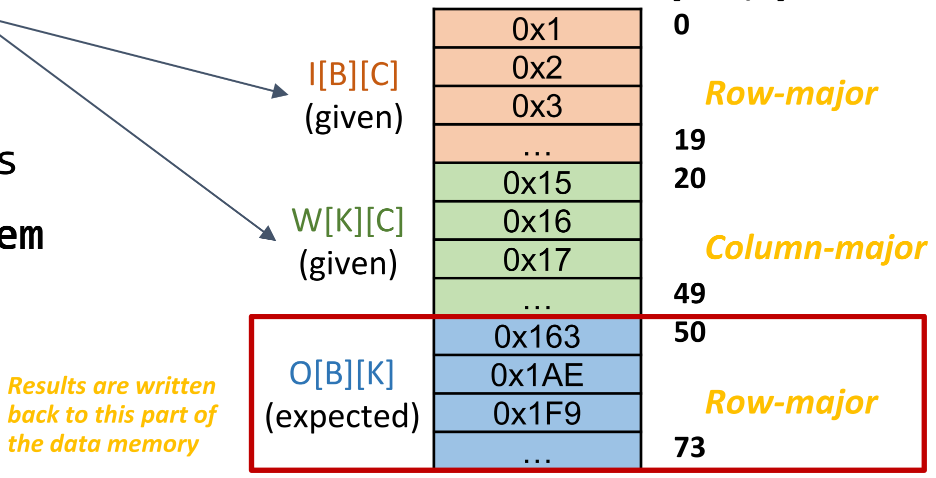
  - Pass the simulation
  - Synthesize

<u>Testing example:</u> (B = 4 , C = 5, K = 6)

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{pmatrix} \cdot \begin{pmatrix} 21 & 26 & 31 & 36 & 41 & 46 \\ 22 & 27 & 32 & 37 & 42 & 47 \\ 23 & 28 & 33 & 38 & 43 & 48 \\ 24 & 29 & 34 & 39 & 44 & 49 \\ 25 & 30 & 35 & 40 & 45 & 50 \end{pmatrix} = \begin{pmatrix} 355 & 430 & 505 & 580 & 655 & 730 \\ 930 & 1130 & 1330 & 1530 & 1730 & 1930 \\ 1505 & 1830 & 2155 & 2480 & 2805 & 3130 \\ 2080 & 2530 & 2980 & 3430 & 3880 & 4330 \end{pmatrix}$$

I[B][C] W[C][K] O[B][K]

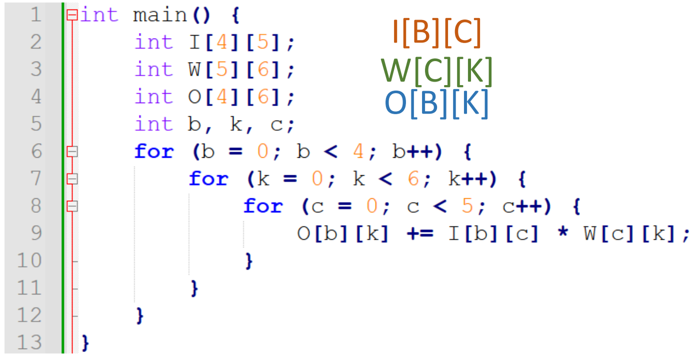
<u>Data memory layout:</u>



We check on this part!

MULT4: our baseline solution

C code:



```c
int main() {
    int I[4][5];
    int W[5][6];
    int O[4][6];
    int b, k, c;
    for (b = 0; b < 4; b++) {
        for (k = 0; k < 6; k++) {
            for (c = 0; c < 5; c++) {
                O[b][k] += I[b][c] * W[c][k];
            }
        }
    }
}
```

I[B][C]
W[C][K]
O[B][K]

Testing example: (B = 4 , C = 5, K = 6)

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{pmatrix} \cdot \begin{pmatrix} 21 & 26 & 31 & 36 & 41 & 46 \\ 22 & 27 & 32 & 37 & 42 & 47 \\ 23 & 28 & 33 & 38 & 43 & 48 \\ 24 & 29 & 34 & 39 & 44 & 49 \\ 25 & 30 & 35 & 40 & 45 & 50 \end{pmatrix} = \begin{pmatrix} 355 & 430 & 505 & 580 & 655 & 730 \\ 930 & 1130 & 1330 & 1530 & 1730 & 1930 \\ 1505 & 1830 & 2155 & 2480 & 2805 & 3130 \\ 2080 & 2530 & 2980 & 3430 & 3880 & 4330 \end{pmatrix}$$

I[B][C] W[C][K] O[B][K]

Required modifications to the processor

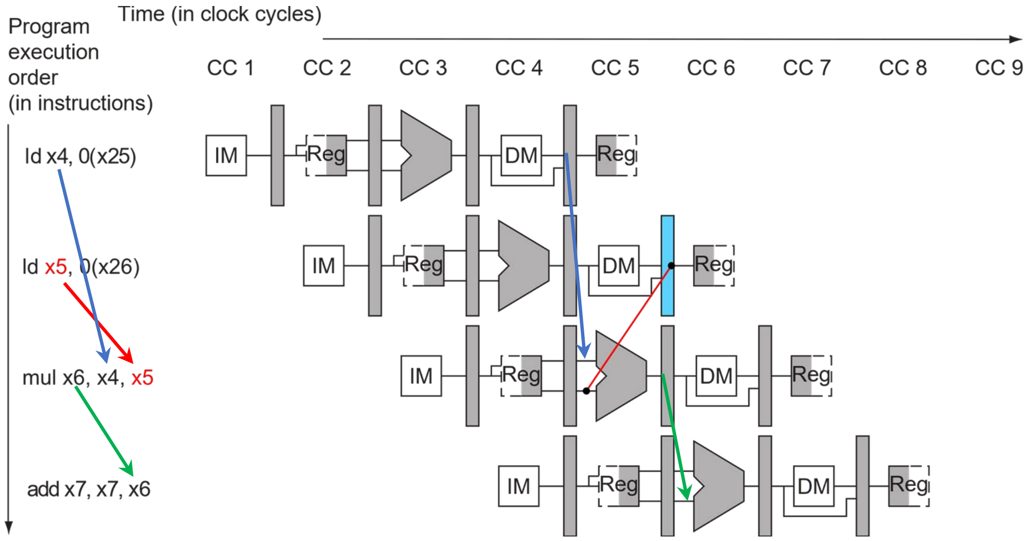
- **Full data forwarding logic**
- **Hazard detection unit (control/data hazard)**

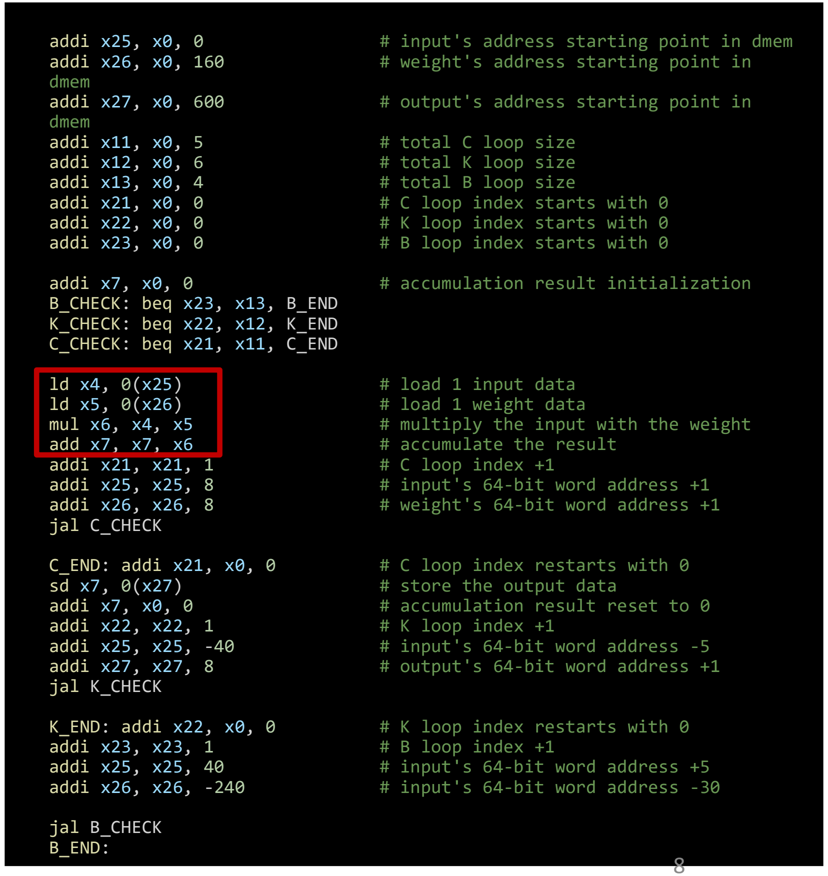
RISC-V assembly code

```
addi x25, x0, 0                      # input's address starting point in dmem
addi x26, x0, 160                    # weight's address starting point in
dmem
addi x27, x0, 600                    # output's address starting point in
dmem
addi x11, x0, 5                      # total C loop size
addi x12, x0, 6                      # total K loop size
addi x13, x0, 4                      # total B loop size
addi x21, x0, 0                      # C loop index starts with 0
addi x22, x0, 0                      # K loop index starts with 0
addi x23, x0, 0                      # B loop index starts with 0

addi x7, x0, 0                       # accumulation result initialization
B_CHECK: beq x23, x13, B_END
K_CHECK: beq x22, x12, K_END
C_CHECK: beq x21, x11, C_END

ld x4, 0(x25)                        # load 1 input data
ld x5, 0(x26)                        # load 1 weight data
mul x6, x4, x5                       # multiply the input with the weight
add x7, x7, x6                       # accumulate the result
addi x21, x21, 1                     # C loop index +1
addi x25, x25, 8                     # input's 64-bit word address +1
addi x26, x26, 8                     # weight's 64-bit word address +1
jal C_CHECK

C_END: addi x21, x0, 0               # C loop index restarts with 0
sd x7, 0(x27)                        # store the output data
addi x7, x0, 0                       # accumulation result reset to 0
addi x22, x22, 1                     # K loop index +1
addi x25, x25, -40                   # input's 64-bit word address -5
addi x27, x27, 8                     # output's 64-bit word address +1
jal K_CHECK

K_END: addi x22, x0, 0               # K loop index restarts with 0
addi x23, x23, 1                     # B loop index +1
addi x25, x25, 40                    # input's 64-bit word address +5
addi x26, x26, -240                  # input's 64-bit word address -30

jal B_CHECK
B_END:
```

I. Full data forwarding logic (Book §4.7)

MULT4:

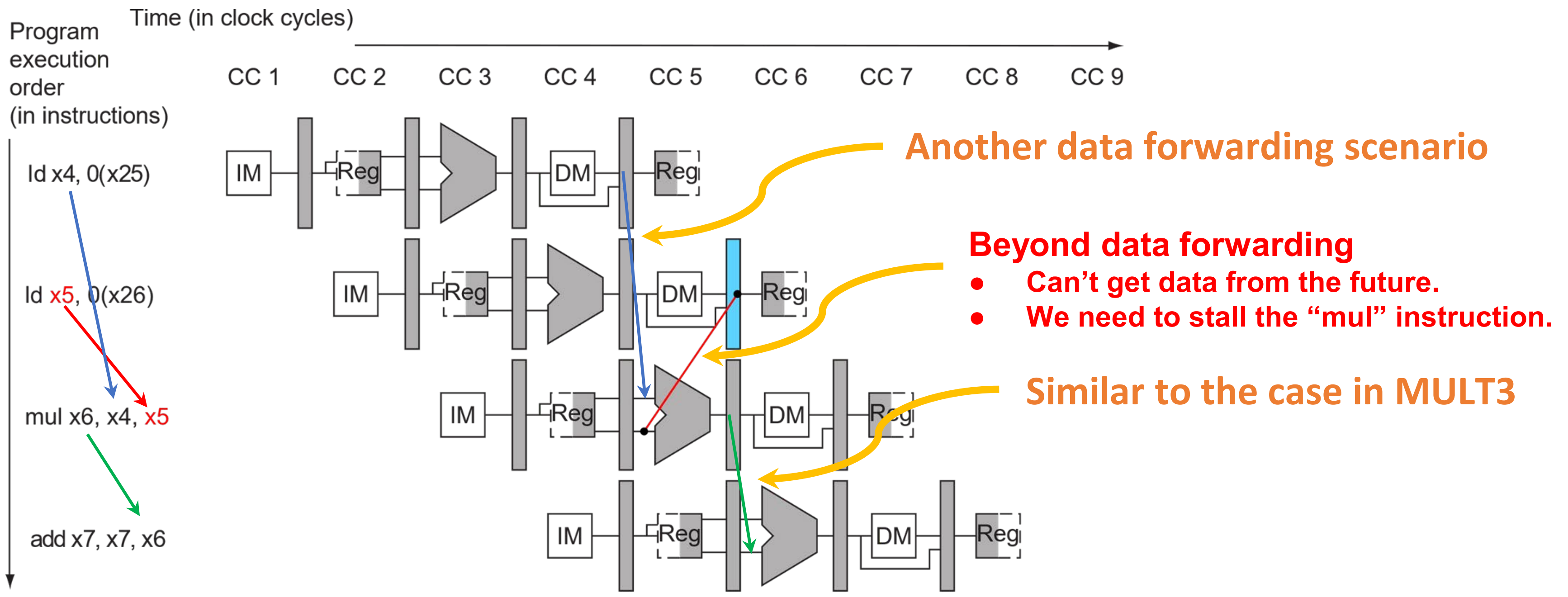




```
addi x25, x0, 0                 # input's address starting point in dmem
addi x26, x0, 160               # weight's address starting point in
dmem
addi x27, x0, 600               # output's address starting point in
dmem
addi x11, x0, 5                 # total C loop size
addi x12, x0, 6                 # total K loop size
addi x13, x0, 4                 # total B loop size
addi x21, x0, 0                 # C loop index starts with 0
addi x22, x0, 0                 # K loop index starts with 0
addi x23, x0, 0                 # B loop index starts with 0

addi x7, x0, 0                  # accumulation result initialization
B_CHECK: beq x23, x13, B_END
K_CHECK: beq x22, x12, K_END
C_CHECK: beq x21, x11, C_END

ld x4, 0(x25)                   # load 1 input data
ld x5, 0(x26)                   # load 1 weight data
mul x6, x4, x5                  # multiply the input with the weight
add x7, x7, x6                  # accumulate the result
addi x21, x21, 1                # C loop index +1
addi x25, x25, 8                # input's 64-bit word address +1
addi x26, x26, 8                # weight's 64-bit word address +1
jal C_CHECK

C_END: addi x21, x0, 0          # C loop index restarts with 0
sd x7, 0(x27)                   # store the output data
addi x7, x0, 0                  # accumulation result reset to 0
addi x22, x22, 1                # K loop index +1
addi x25, x25, -40              # input's 64-bit word address -5
addi x27, x27, 8                # output's 64-bit word address +1
jal K_CHECK

K_END: addi x22, x0, 0          # K loop index restarts with 0
addi x23, x23, 1                # B loop index +1
addi x25, x25, 40               # input's 64-bit word address +5
addi x26, x26, -240             # input's 64-bit word address -30

jal B_CHECK
B_END:
```

I. Full data forwarding logic (Book §4.7)

MULT4:

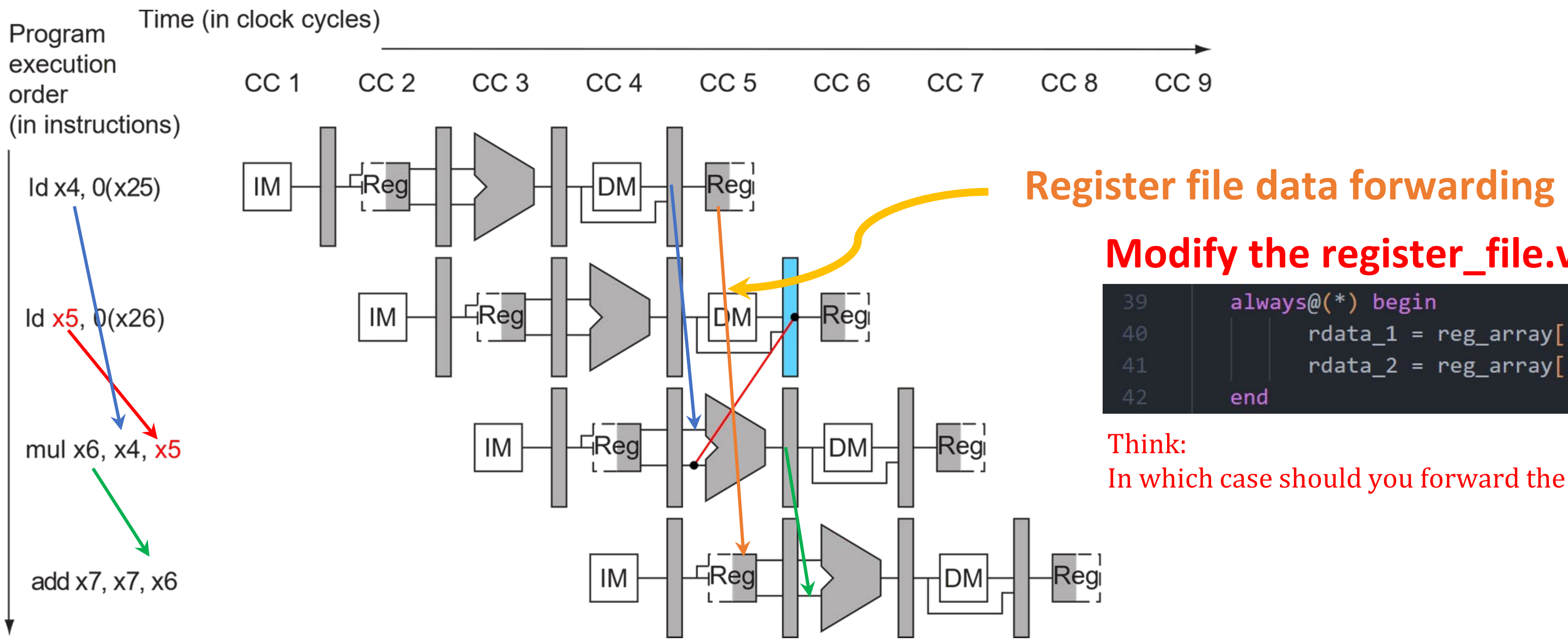


Beyond data forwarding

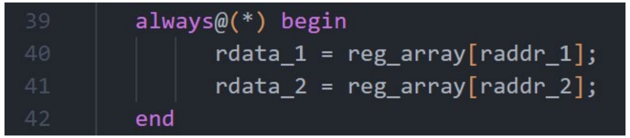
- **Can't get data from the future.**
- **We need to stall the "mul" instruction.**

I. Full data forwarding logic (Book §4.7)

MULT4:



Modify the register_file.v

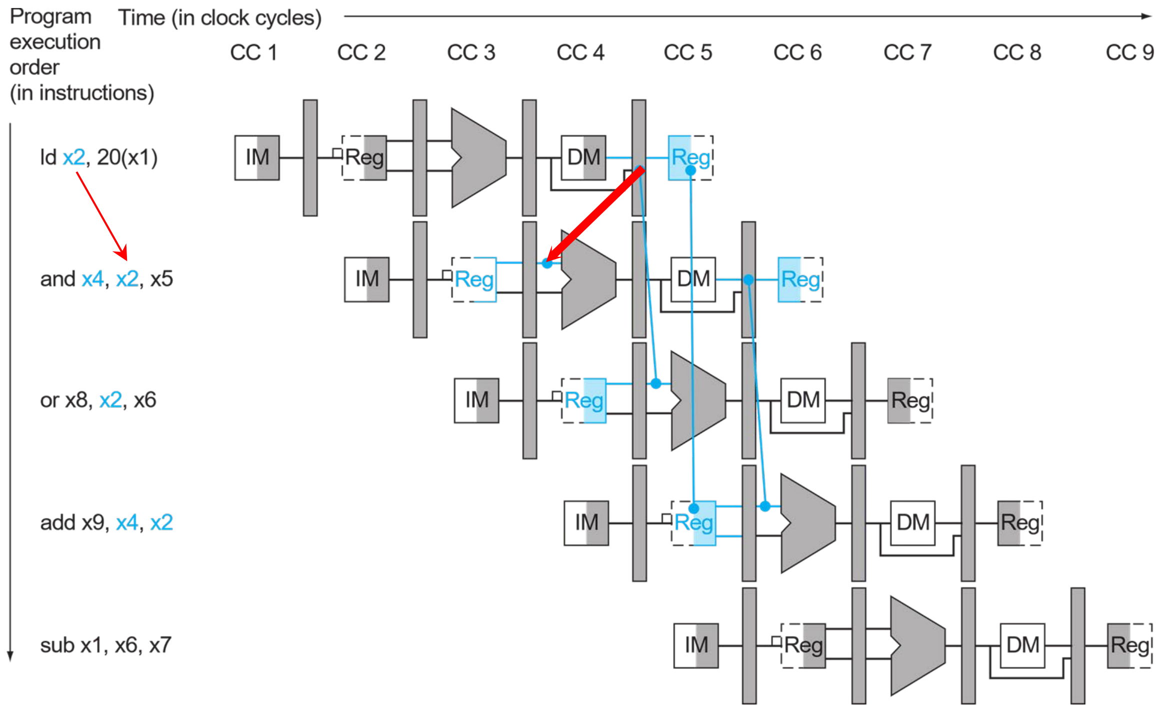


```
always@(*) begin
        rdata_1 = reg_array[raddr_1];
        rdata_2 = reg_array[raddr_2];
end
```

Think:
In which case should you forward the **wdata** to **rdata**?

II. Data hazard resolution with stalling

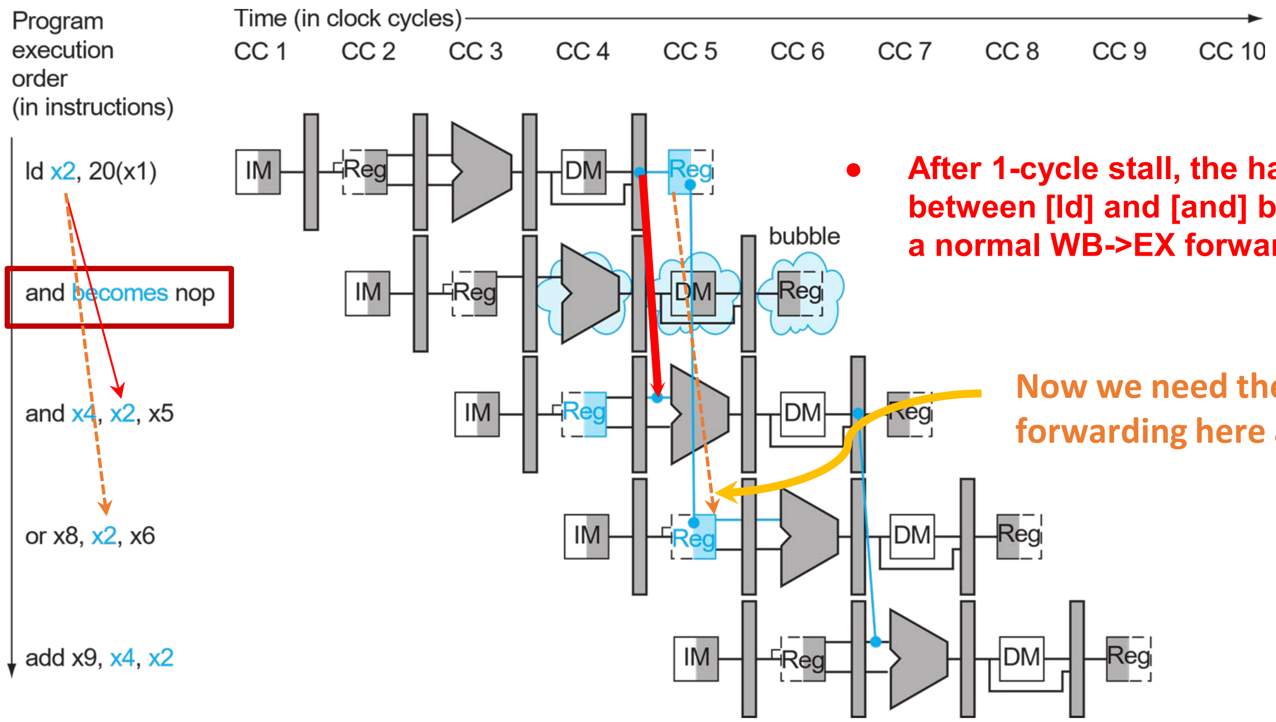
Load and use



Patterson Book. FIGURE 4.56

II. Data hazard resolution with stalling

Load and use



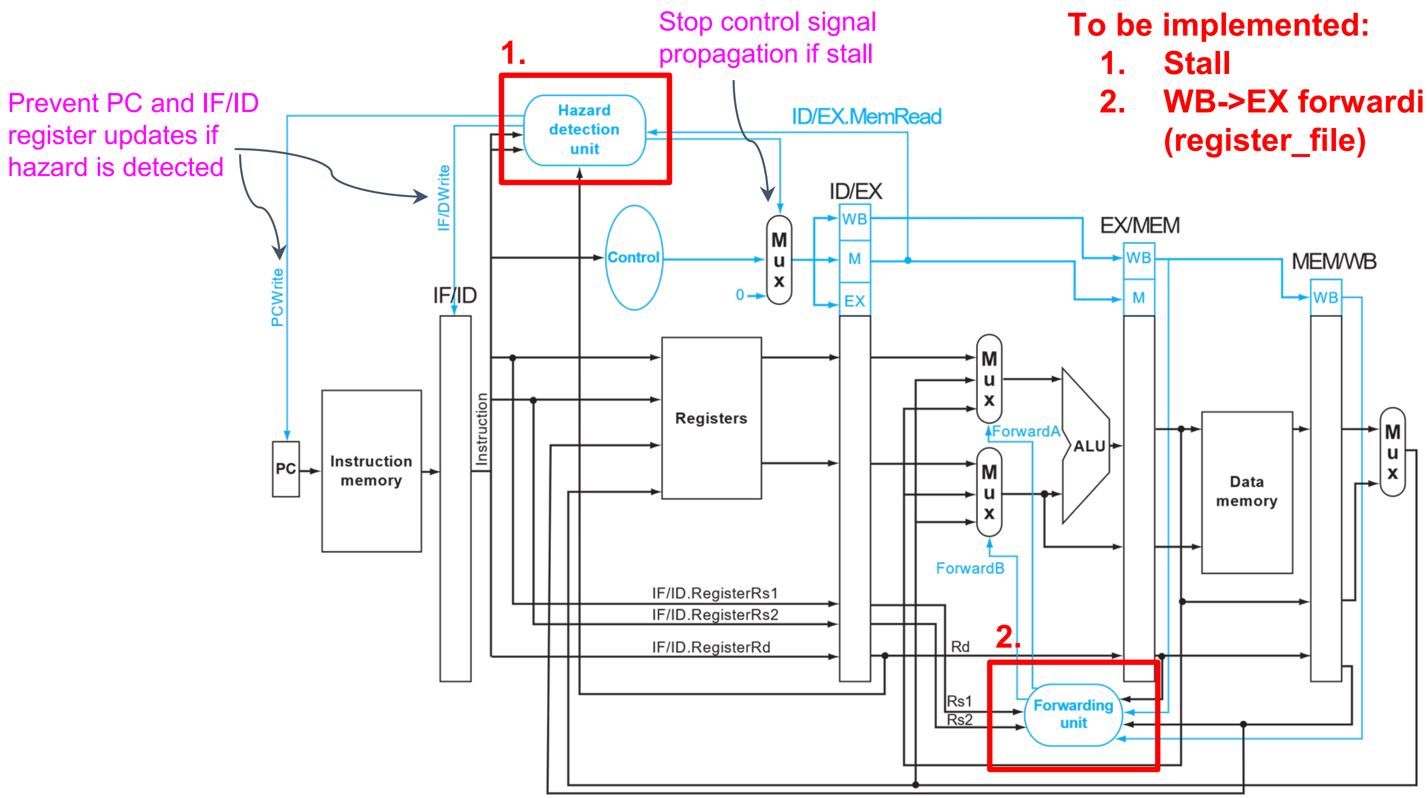
- **After 1-cycle stall, the hazard between [ld] and [and] becomes a normal WB->EX forwarding.**

Now we need the register file forwarding here after stalling.

II. Data hazard resolution with stalling

To be implemented:

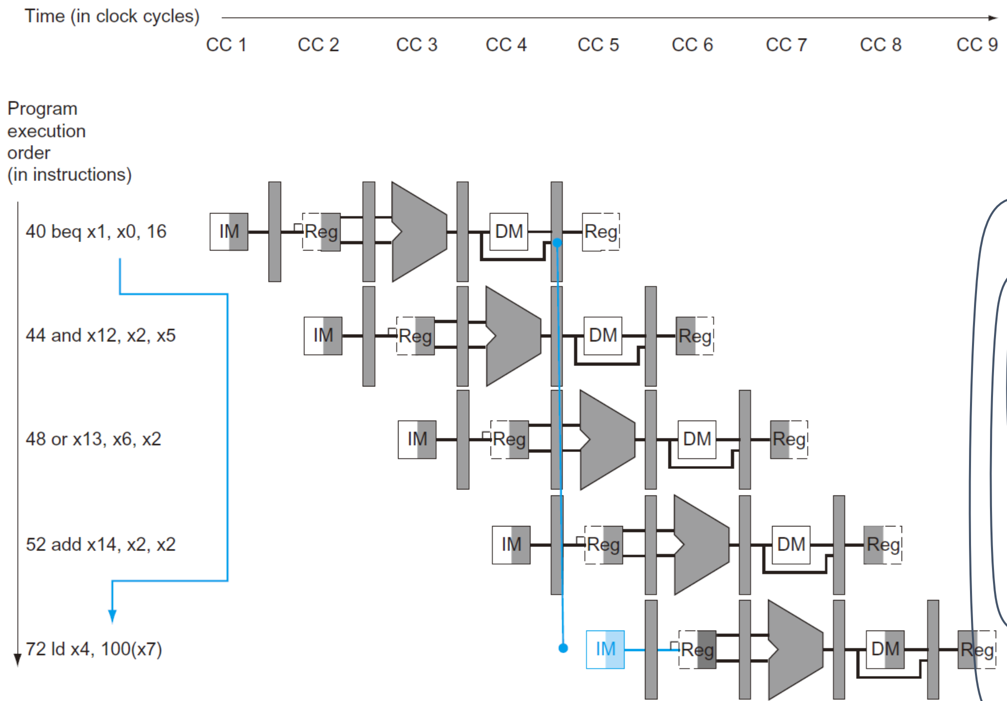
1. **Stall**
2. **WB->EX forwarding (register_file)**

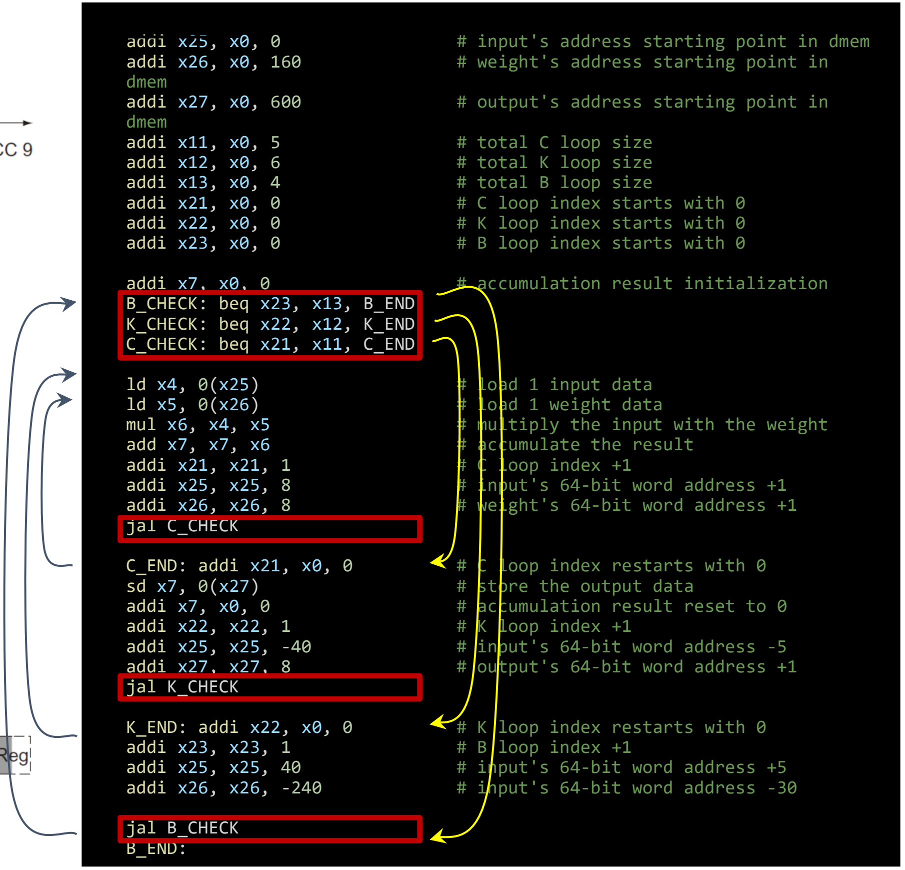


Patterson Book. FIGURE 4.58

III. Control Hazard solution (Book §4.8)

Control Hazard:



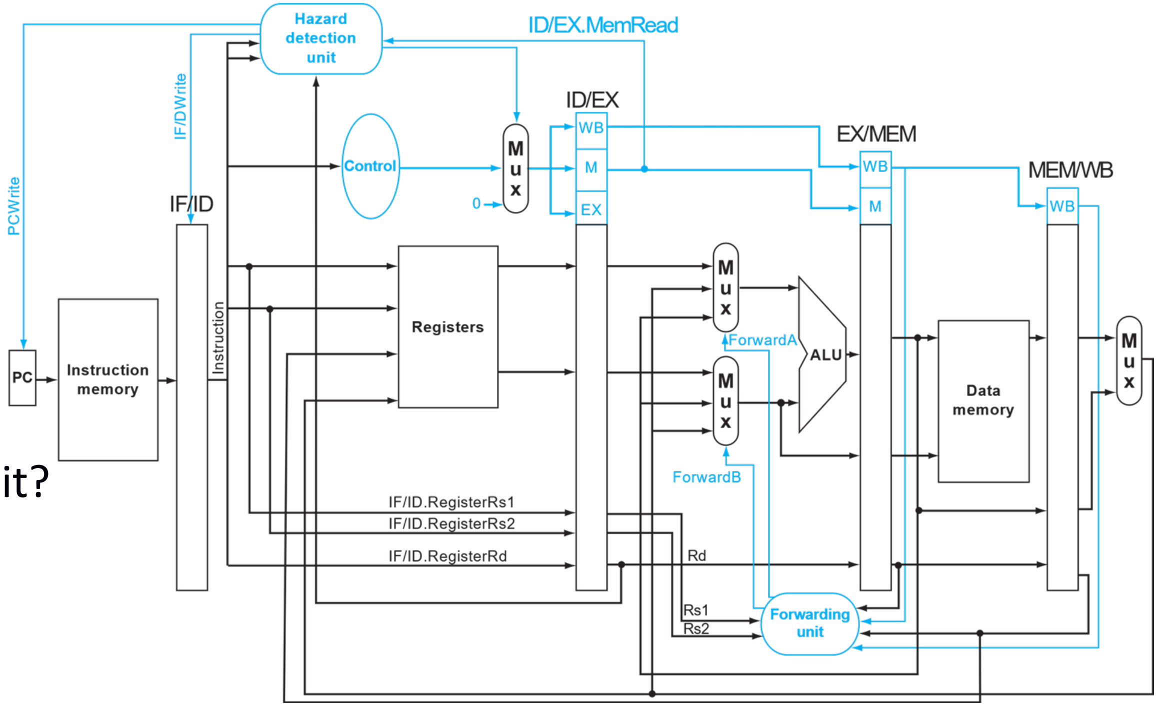


```
addi x25, x0, 0                # input's address starting point in dmem
addi x26, x0, 160              # weight's address starting point in
dmem
addi x27, x0, 600              # output's address starting point in
dmem
addi x11, x0, 5                # total C loop size
addi x12, x0, 6                # total K loop size
addi x13, x0, 4                # total B loop size
addi x21, x0, 0                # C loop index starts with 0
addi x22, x0, 0                # K loop index starts with 0
addi x23, x0, 0                # B loop index starts with 0

addi x7, x0, 0                 # accumulation result initialization
B_CHECK: beq x23, x13, B_END
K_CHECK: beq x22, x12, K_END
C_CHECK: beq x21, x11, C_END

ld x4, 0(x25)                  # load 1 input data
ld x5, 0(x26)                  # load 1 weight data
mul x6, x4, x5                 # multiply the input with the weight
add x7, x7, x6                 # accumulate the result
addi x21, x21, 1               # C loop index +1
addi x25, x25, 8               # input's 64-bit word address +1
addi x26, x26, 8               # weight's 64-bit word address +1
jal C_CHECK

C_END: addi x21, x0, 0         # C loop index restarts with 0
sd x7, 0(x27)                  # store the output data
addi x7, x0, 0                 # accumulation result reset to 0
addi x22, x22, 1               # K loop index +1
addi x25, x25, -40             # input's 64-bit word address -5
addi x27, x27, 8               # output's 64-bit word address +1
jal K_CHECK

K_END: addi x22, x0, 0         # K loop index restarts with 0
addi x23, x23, 1               # B loop index +1
addi x25, x25, 40              # input's 64-bit word address +5
addi x26, x26, -240            # input's 64-bit word address -30

jal B_CHECK
B_END:
```

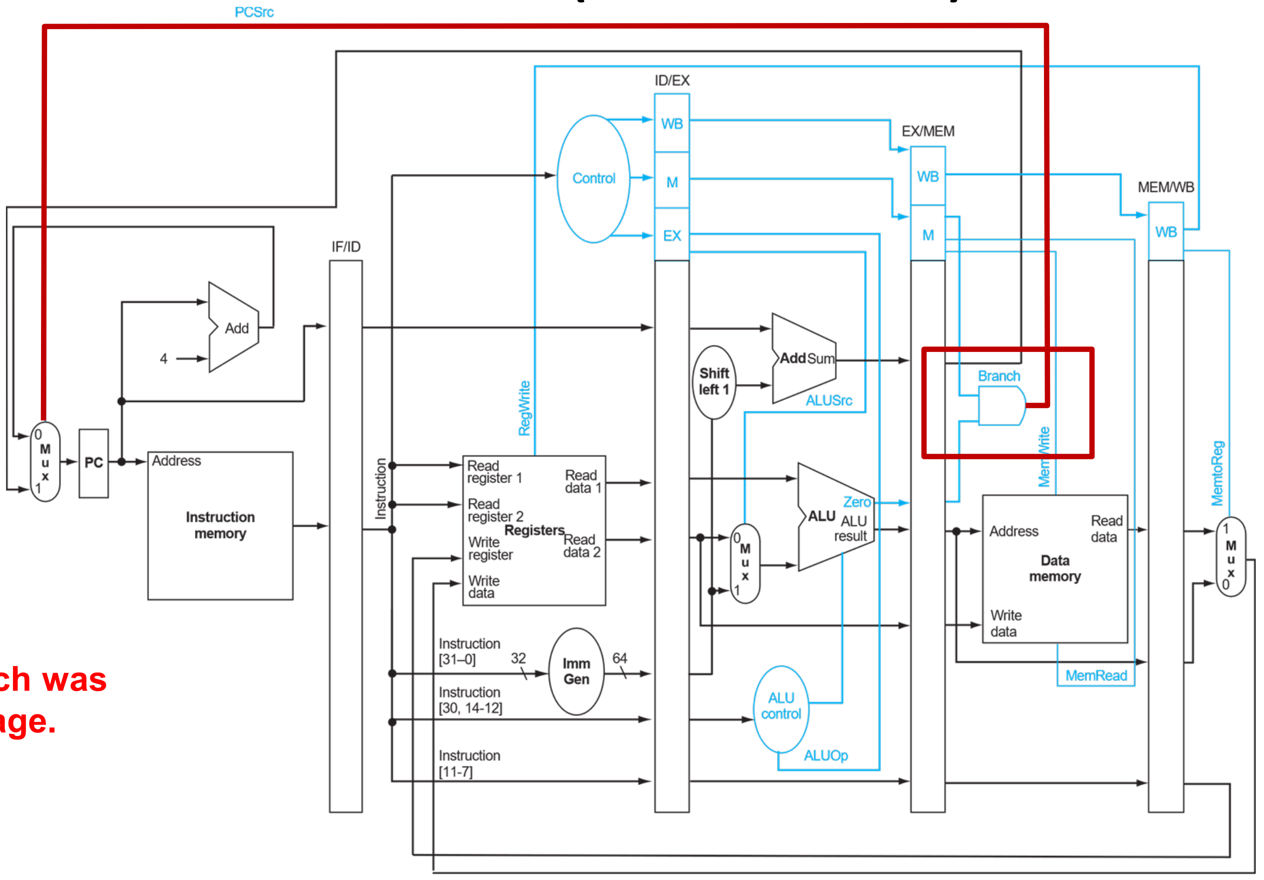
III. Control Hazard solution (Book §4.8)

Branch Unit ?

- We missed our Branch Unit!
  - Not used in MULT2 & MULT3
  - But will be useful here for
    - **BR** instruction
    - **JAL** instuction
  - Which pipeline should we put it?
    - What is its input?
    - How is the data computed?
    - Who is using its output?



III. Control Hazard solution (Book §4.8)

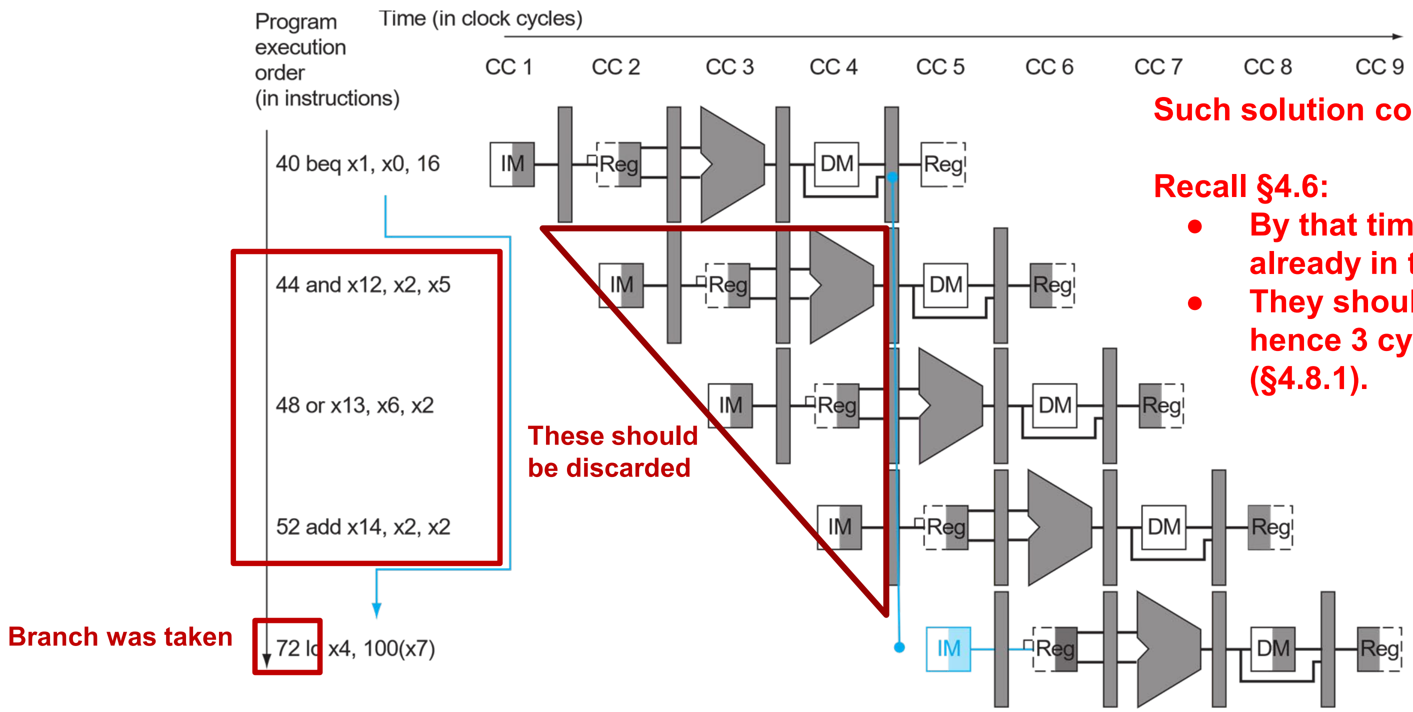


Recall §4.6:

- **We know if the branch was taken at the MEM stage.**

Patterson Book. FIGURE 4.49

III. Control Hazard solution (Book §4.8)

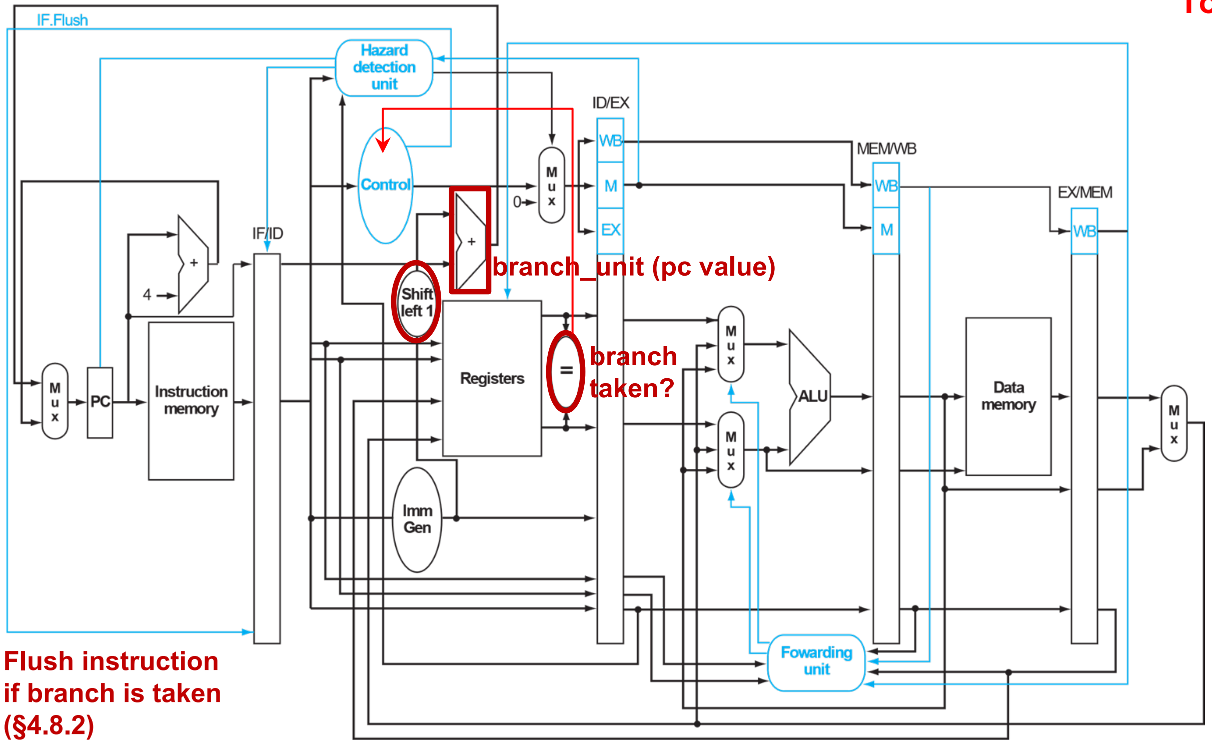


Such solution comes at a cost…

Recall §4.6:

- **By that time, 3 instructions are already in the pipeline.**
- **They should be discarded, hence 3 cycles are wasted (§4.8.1).**

Control Hazard - Our baseline



Flush instruction if branch is taken (§4.8.2)

Patterson Book. FIGURE 4.62

To save this overhead:

- **Make branch/jump decision *ASAP* (at the ID stage, §4.8.2).**
- **This is our baseline architecture for MULT4 (in terms of saving the total cycle amount).**
- **We still use the branch-not-taken prediction (in our baseline).**
  a. **ID judges the branch/jump.**
  b. **IF basically fetches imem[PC+4].**
  c. **If branch-taken/jump, next PC is overwritten and the false IF instruction is flushed before next clock edge comes.**
- **Feel free to use different prediction scheme if you find it valuable(§4.8.3).**

IV. PLAN B – hazards can be solved with nop insertion

Recall **MULT2**, the software solution.

[Fail-Safe] If the deadline is right ahead, at least make your processor functional on MULT4!

```
addi x25, x0, 0             # input's address starting point in dmem
addi x26, x0, 160           # weight's address starting point in dmem
addi x27, x0, 600           # output's address starting point in dmem
addi x11, x0, 5             # total C loop size
addi x12, x0, 6             # total K loop size
addi x13, x0, 4             # total B loop size
addi x21, x0, 0             # C loop index starts with 0
addi x22, x0, 0             # K loop index starts with 0
addi x23, x0, 0             # B loop index starts with 0

addi x7, x0, 0              # accumulation result initialization
B_CHECK: beq x23, x13, B_END
nop
nop
nop
K_CHECK: beq x22, x12, K_END
nop
nop
nop
C_CHECK: beq x21, x11, C_END
nop
nop
nop
```

- **Don't just copy and paste this one!**
- **Understand your architecture and insert the necessary nops.**
- **Hazard control and nop insertion are related**
  - **The better hazard control logic you have,**
  - **The fewer nops you will need,**
  - **The better performance you will achieve.**

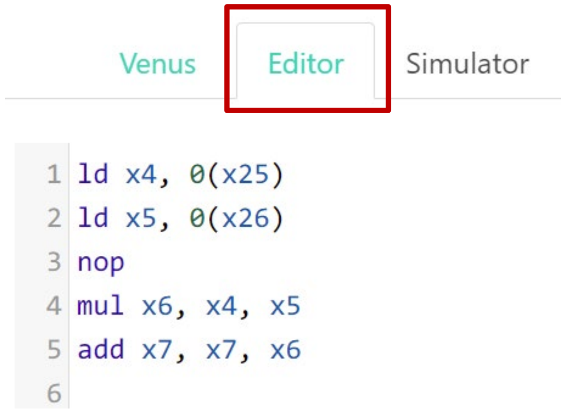
Understand the program, locate the hazard, understand your design, upgrade the imem…

```
ld x4, 0(x25)               # load 1 input data
ld x5, 0(x26)               # load 1 weight data
nop
mul x6, x4, x5              # multiply the input with the weight
add x7, x7, x6              # accumulate the result
addi x21, x21, 1            # C loop index +1
addi x25, x25, 8            # input's 64-bit word address +1
addi x26, x26, 8            # weight's 64-bit word address +1
jal C_CHECK
nop
nop
nop

C_END: addi x21, x0, 0      # C loop index restarts with 0
sd x7, 0(x27)               # store the output data
addi x7, x0, 0              # accumulation result reset to 0
addi x22, x22, 1            # K loop index +1
addi x25, x25, -40          # input's 64-bit word address -5
addi x27, x27, 8            # output's 64-bit word address +1
jal K_CHECK
nop
nop
nop


K_END: addi x22, x0, 0      # K loop index restarts with 0
addi x23, x23, 1            # B loop index +1
addi x25, x25, 40           # input's 64-bit word address +5
addi x26, x26, -240         # input's 64-bit word address -30

jal B_CHECK
nop
nop
nop
B_END:
```

IV. PLAN B – hazards can be solved with nop insertion

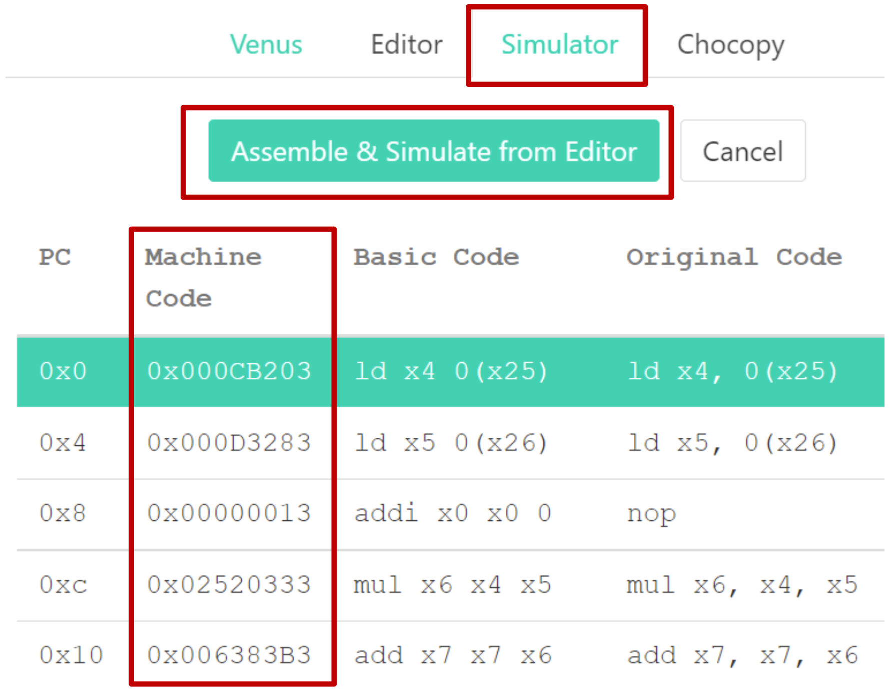
- Whenever you change the program, DO NOT FORGET to regenerate the machine code (imem_content)!
- Make use of the online tool we mentioned in session-1 (link-online-converter, link-github).

1. Paste your assembly code in Editor.



```
ld x4, 0(x25)
ld x5, 0(x26)
nop
mul x6, x4, x5
add x7, x7, x6
```

2. Go to Simulator and click the Green button.



PC	Machine Code	Basic Code	Original Code
0x0	0x000CB203	ld x4 0(x25)	ld x4, 0(x25)
0x4	0x000D3283	ld x5 0(x26)	ld x5, 0(x26)
0x8	0x00000013	addi x0 x0 0	nop
0xc	0x02520333	mul x6 x4 x5	mul x6, x4, x5
0x10	0x006383B3	add x7 x7 x6	add x7, x7, x6

3. Upgrade the imem_content (remove "0x"!).
4. Never forget to add the finishing lines. 👇

```
FA9FF0EF // jal B_CHECK
00000013
00000013
00000013
00000013
4000007E // STOP instruction mult4
```

Today’s session: task summary

With **session_guide.pdf**

- Study the RUN CYCLE-ACCURATE SIMULATION and RUN BACKEND FLOW
- Follow the TASKS TO BE DONE and fill in the **report.docx**

Copy-paste your finished **/RTL/*.v** into the SOLUTION folders.

- **Obj-1** → **RTL_SOLUTION5_pipeline_hazard_advanced_MULT4**

- **Note:**

1. We use universal test patterns for fair grading.
2. **Do not modify cpu_tb.v & sky130_sram_2rw.v**
3. **Do not modify mult4_dmem_content.txt**
4. **This time you can modify mult4_imem_content.txt**

Be creative

- Useful resources to make it go faster!
  - Patterson Book. End of section 4.8 on branch prediction tricks.
  - Patterson Book. From Section 4.10 on advanced acceleration tricks.
  - RISC-V specification. Chapter 17 on the “V” Standard Extension for Vector Operations

Grading

ITEM	Points
Functional pipelined MULT2	0.5
Functional pipelined MULT3	0.5
Functional MULT4	0.4
Functional MULT4 #cycles ≤ baseline impl. (1636 cc)	0.8
Advanced MULT4 #cycles ≤ advanced impl. (828 cc)	0.8
Report	1.0
Total	**4.0**

Project handover

Deadline: **May 2nd**